About raising and handling exceptions

Dominique Duval and Jean-Claude Reynaud

April 20., 2006

Abstract. This paper presents a unified framework for dealing with a deduction system and a denotational semantics of exceptions. It is based on the fact that handling exceptions can be seen as a kind of generalized case distinction. This point of view on exceptions has been introduced in 2004, it is based on the notion of diagrammatic logic, which assumes some familiarity with category theory. Extensive sums of types can be used for dealing with case distinctions. The aim of this new paper is to focus on the role of a generalized extensivity property for dealing with exceptions. Moreover, the presentation of this paper makes only a restricted use of category theory.

Keywords: semantics of exceptions, case distinction, extensive sums, diagrammatic logic.

1 Introduction

This paper presents a unified framework for dealing with a deduction system and a denotational semantics of exceptions. It is based on the fact that handling exceptions can be seen as a kind of generalized case distinction. This point of view on exceptions has been introduced in [6], and a short presentation can be found in [7]. In both these papers, some familiarity with category theory (adjunction, sketches,...) is assumed. One aim of this new paper to present the main ideas of [6] in an elementary way, with a restricted use of category theory.

Usual case distinction can be presented in a *distributive* logic, which means that products and sums of types are allowed, and that the product is distributive over the sum. Products and sums of types can be interpreted as cartesian products and disjoint unions of sets, respectively, so that the distributivity property does hold on sets. It follows from [3] that case distinction can also be presented in a weaker *extensive* logic, where sums of types are allowed, and the inverse image of a sum by a function is still a sum. In this paper, exceptions are formalized in a kind of generalized extensive logic; in [6], this framework is enriched for dealing also with product types. Exceptions are studied in many different frameworks, for instance in [8, 2, 14, 1, 9, 13, 15]. But, to our knowledge, the emphasize on the use of the extensivity property for dealing with exceptions, is new.

A puzzling issue about exceptions is the apparent discrepancy between the deduction system of a language with exceptions and its set-valued interpretation. Indeed, the type of exceptions is implicit in the language, while its interpretation requires an explicit set of exceptions. A major step towards a solution is the use of *monads* in [11], in the framework of typed lambda-calculus: the functions are classified, on the one hand the *values* are not allowed to raise any exception, on the other hand the *computations* may raise an exception. So, if the types X and Y are interpreted in a set-valued model as the sets A and B, then a function $f : X \to Y$ is interpreted either as a map $\varphi : A \to B$ if f is a value, or as a map $\varphi : A \to B + \mathbb{E}$, where $\mathbb{E}$ is the set of exceptional values, if f is a computation. But this approach fails to formalize in a satisfactory way the handling of exceptions in the framework of typed lambda-calculus [12]. Our approach succeeds in formalizing the handling of exceptions, but the extensive logic is fairly different from typed lambda-calculus. Although we do not use monads explicitly, we do distinguish values from computations.

Actually, three different extensive logics are presented in this paper. The *basic* extensive logic is described in section 3: there are sums of types, and the inverse image of a sum by a function is a sum. This basic logic does not deal with exceptions. In the next sections, it is modified in two different ways, in order to include a treatment of exceptions. The *decorated logic with exceptions*, or simply *decorated logic*, is described in section 4. Then the *logic with explicit exceptions*, or simply *explicit logic*, is presented in section 5. Each of both logics for exceptions has its own deduction system and denotational semantics, however the interest of the first one relies primarily in its deduction system, while the denotational semantics of the second one is easier to grasp. A link between these logics is established, so that the deduction system of the decorated logic is sound with respect to the models in the sense of the explicit logic. This solves the problem of the apparent discrepancy between the deduction system of a language with exceptions and its set-valued interpretation.

So, this point of view on exceptions requires a framework for dealing with several logics and the links between them. Such a framework is provided by *diagrammatic logics* [5, 4]. This work does rely on the theory of diagrammatic logics, mainly for the definition of the decorated logic and for the link between the decorated logic and the explicit logic, as explained in [6, 7]. However, in this paper, the role of diagrammatic logic is hidden, and the few required notions about categories are reminded. Actually, we do not need much more than the definition of a category, which is quite simple: it is a directed graph where the arrows can be composed as soon as they are consecutive. Proofs can be found in [7].

A diagrammatic logic is well known as soon as its *specifications* and *theories* are carefully described. Roughly speaking, a specification is a family of axioms, and a theory is a family of theorems that is closed under deduction. The *deduction rules* of the given diagrammatic logic are used for generating a theory from a specification, which means, for deriving theorems from axioms. The *models* of a specification are then defined automatically, in a sound way: every theorem that can be proved from a specification is satisfied in every model of the specification, or equivalently, every model of the specification can be extended to a model of the generated theory.

2 About graphs

In the three logics that will be described, the specifications and theories are some kind of generalized graphs and categories, respectively. In this preliminary section, we introduce some basic facts about graphs and categories, that will be used in the next sections.

Definition 2.1 (graph). A *(directed multi-)graph* is made of points and arrows, that are called respectively *types* $X, Y, \ldots$ and *(univariate) functions* $f : X \to Y, \ldots$

A *category* is a graph where functions can be composed, with the usual properties of composition, as follows.

Definition 2.2 (category). A *category* is a graph where each type has an *identity* function $\mathtt{id}_X : X \to X$, each pair of consecutive functions $f : X \to Y$ and $g : Y \to Z$ has a *composed* function $g \,.\, f : X \to Z$, and the *unitarity* and *associativity* axioms hold (as soon as it makes sense):

$$f \,.\, \mathtt{id}_X = f \ , \ \mathtt{id}_Y \,.\, f = f \ , \ (h \,.\, g) \,.\, f = h \,.\, (g \,.\, f) \ .$$

As usual, thanks to associativity, parentheses are generally dropped.

Clearly, each graph generates a category, by adding all the missing identities and composed functions, and by identifying some functions according to the axioms. Generating a category from a graph is similar to generating *all* the programs from a grammar of a given language, or generating *all* the theorems about groups (say) from a set of axioms for groups. This is pretty interesting, but far too large: we

are usually quite happy with *some* programs and *some* theorems... More is said about this remark in the "decomposition theorem" of [5, 4]. About graphs and categories, this remark is the motivation for defining something "between" both, as follows.

Definition 2.3 (compositive graph). A *compositive graph* is a graph where each type may have a (potential) identity function $\mathtt{id}_X : X \to X$ and each pair of consecutive functions $f : X \to Y$ and $g : Y \to Z$ may have a (potential) composed function $g \,.\, f : X \to Z$.

The unitarity and associativity axioms are not mentioned: as any equalities, some of them may hold, but this is not mandatory. Typically, a compositive graph may describe a step between a graph and its generated category, when *some* identities and composed functions have been generated.

The compositive graphs and the categories form the specifications and theories, respectively, of a (very simple) diagrammatic logic. The rules of this logic are the identity and composition rules, as well as the rules that correspond to the axioms for categories:

$$\frac{X}{\mathtt{id}_X : X \to X}\ (\mathrm{id}) \qquad \frac{f : X \to Y \quad g : Y \to Z}{g \,.\, f : X \to Z}\ (\mathrm{comp})$$

$$\frac{f : X \to Y}{f \,.\, \mathtt{id}_X = f : X \to Y}\ (\mathrm{unit}_X) \qquad \frac{f : X \to Y}{\mathtt{id}_Y \,.\, f = f : X \to Y}\ (\mathrm{unit}_Y)$$

$$\frac{f : X \to Y \quad g : Y \to Z \quad h : Z \to T}{(h \,.\, g) \,.\, f = h \,.\, (g \,.\, f) : X \to T}\ (\mathrm{assoc})$$

The fact that types and functions can be considered as symbols that stand for sets and maps, respectively, is catched by the following notion of *model*. In this paper, only set-valued models are considered; a more general definition of models can be found in [5, 4]. For clarity, we speak about *maps* (rather than functions) between sets.

Definition 2.4 (model of a compositive graph). A *(set-valued) model* M of a compositive graph interprets each type X as a set $M(X)$ and each function $f : X \to Y$ as a map $M(f) : M(X) \to M(Y)$, in such a way that identity functions are interpreted as identity maps and composed functions as composed maps: $M(\mathtt{id}_X) = \mathtt{id}_{M(X)}$ and $M(g \,.\, f) = M(g) \,.\, M(f)$.

Example 2.5 (natural numbers). Let us consider the graph made of two types Unit and Nat and two functions $z : \mathrm{Unit} \to \mathrm{Nat}$ and $s : \mathrm{Nat} \to \mathrm{Nat}$:

$$\mathrm{Unit} \xrightarrow{\ z\ } \mathrm{Nat} \circlearrowright s$$

The generated category contains the functions $\mathtt{id}_{\mathrm{Unit}}$, $\mathtt{id}_{\mathrm{Nat}}$, as well as $s^k : \mathrm{Nat} \to \mathrm{Nat}$ and $s^k \,.\, z : \mathrm{Unit} \to \mathrm{Nat}$ for every $k \in \mathbb{N}$. By adding to the initial graph some of these functions, we get a compositive graph. The *model of naturals* of all these graphs interprets Unit as a singleton $\{*\}$, Nat as the set $\mathbb{N}$ of naturals, z as the constant map $* \mapsto 0$, which is identified to the element $0 \in \mathbb{N}$, and s as the successor map $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$. Then the function $s^k \,.\, z$ is interpreted as the constant map $* \mapsto k$, identified to $k \in \mathbb{N}$.

There is still a technical point to discuss about compositive graphs and categories. Equality between functions is often too crude for dealing with computational issues: for a compiler, functions like $f \,.\, \mathtt{id}_X$ and f are distinct, even though they become identified in all models. This is a reason for introducing *equations* $f \equiv g$ as *potential equalities* in compositive graphs: if $f \equiv g$, then $M(f) = M(g)$ in every model M. So, from now on, every compositive graph may have equations.

It follows that the categories also have to be modified. An *equiv-category* looks like a category, except for two points. First, it is equipped with equations which form a *congruence*, which means, an equivalence relation compatible with composition. Second, it satisfies the unitarity and associativity axioms only up

to congruence. For simplicity, and because this will not cause any trouble in this paper, we still call it a *category*.

So, this diagrammatic logic is a kind of equational logic, where all functions have arity 1.

3 A basic logic

In order to focus on the issue of exceptions, we have chosen a *basic logic* that deals with case distinctions. As in section 2, all its functions have arity 1, since no product of types is provided; multivariate functions are considered in [6]. In order to deal with case distinctions, some sums of types are needed, and they must satisfy a property called *extensivity*, after [3]. Note that in [3] the word "extensivity" is used only for categories, while here it is used for sums. The specifications and theories of the basic logic are described below.

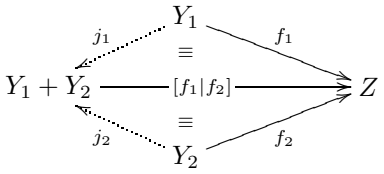
Definition 3.1 (basic specification). A *basic specification* Σ is a compositive graph such that some finite lists of types $Y_1, \ldots, Y_n$ have a *(potential) sum*, made of a *vertex* type $Y_1 + \ldots + Y_n$ and *coprojection* functions $j_i : Y_i \to Y_1 + \ldots + Y_n$, for $i \in \{1, \ldots, n\}$.

Definition 3.2 (models of a basic specification). A *(set-valued) model* of a basic specification is a model of the underlying compositive graph that interprets potential sums as disjoint unions.

The properties of sums in a basic theory are stated now. The first one (existence and unicity of matches) is the usual defining property of sums in a category, but only up to congrunce. The second property (extensivity of sums) will allow to define case distinction.

Definition 3.3 (sums and matches). A *sum* is a potential sum that satisfies the following property. If $f_i : Y_i \to Z$, for $i \in \{1, \ldots, n\}$, are functions, then there is a *match* $[j_1 \Rightarrow f_1 \mid \ldots \mid j_n \Rightarrow f_n]$ or $[f_1 \mid \ldots \mid f_n] : Y_1 + \ldots + Y_n \to Z$, i.e., a function such that $[f_1 \mid \ldots \mid f_n] \,.\, j_i \equiv f_i$ for $i \in \{1, \ldots, n\}$, and if $f : Y_1 + \ldots + Y_n \to Z$ is a function such that $f \,.\, j_i \equiv f_i$ for $i \in \{1, \ldots, n\}$ then $f \equiv [f_1 \mid \ldots \mid f_n]$.

The existence of matches can be illustrated as follows, when $n = 2$, with dotted arrows for representing the coprojections:



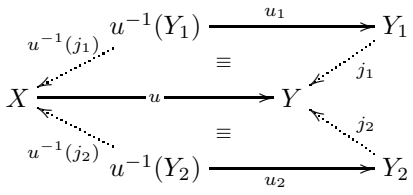
When $n = 0$, a sum "of no type" is called an *initial* type, denoted 0; it satisfies the following property. If Z is a type, then there is a function $[\,]_Z : 0 \to Z$ such that, if $f : 0 \to Z$ is a function, then $f \equiv [\,]_Z$. The existence of empty matches can be illustrated as follows:

$$0 \longrightarrow[\,]_Z\longrightarrow Z$$

Definition 3.4 (the inverse image of a sum by a function). Let $Y = Y_1 + \ldots + Y_n$ be a sum, with coprojections $j_1, \ldots, j_n$, and let $u : X \to Y$ be a function. An *inverse image of the sum* $Y = Y_1 + \ldots + Y_n$ *by the function* u is a sum $X = u^{-1}(Y_1) + \ldots + u^{-1}(Y_n)$, with coprojections $u^{-1}(j_1), \ldots, u^{-1}(j_n)$, together with *restriction* functions $u_i : u^{-1}(Y_i) \to Y_i$ such that, for $i \in \{1, \ldots, n\}$:

$$j_i \,.\, u_i \equiv u \,.\, u^{-1}(j_i) \,.$$

Here is an illustration when $n = 2$.



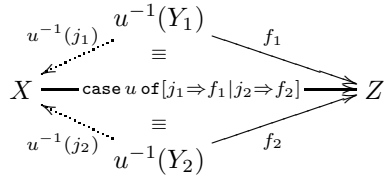
Definition 3.5 (extensivity). A sum $Y = Y_1 + \ldots + Y_n$ is *extensive* if, for every function $u : X \to Y$ there is an inverse image of the sum $Y = Y_1 + \ldots + Y_n$ by the function u, and it is unique (the unicity of inverse images, here and in the sequel, is only up to some equivalence).

Definition 3.6 (basic theories). A *basic theory* Θ is a basic specification such that its underlying graph is a category, and all its potential sums of types are extensive sums.

The category of sets can be seen as a basic theory, with the equality for congruence. It is not assumed here that all sums of types do exist in a basic theory, although this property could be added. Now, case distinction in any basic theory is easily defined, thanks to the properties of sums.

Definition 3.7 (cases). Let $Y = Y_1 + \ldots + Y_n$ be a sum, $u : X \to Y$ a function, and let $X = u^{-1}(Y_1) + \ldots + u^{-1}(Y_n)$ be the inverse image. Let $f_i : u^{-1}(Y_i) \to Z$ be functions, for $i \in \{1, \ldots, n\}$. The *case distinction* function (or simply the *case* function) that acts as f_i on $u^{-1}(Y_i)$, for all i, is:

$$\texttt{case}\ u\ \texttt{of}\ [\, j_i \Rightarrow f_i \,]_{1 \le i \le n} \ = \ [\, u^{-1}(j_i) \Rightarrow f_i \,]_{1 \le i \le n} \ : \ X \to Z \ .$$



This means that the case function is characterized by the equations:

$$(\texttt{case}\ u\ \texttt{of}\ [\, j_i \Rightarrow f_i \,]_{1 \le i \le n}) \,.\, (u^{-1}(j_i)) \equiv f_i \quad \text{, for } \ 1 \le i \le n \ .$$

Clearly, when $u = \texttt{id}_Y : Y \to Y$, then the case function is congruent to a match:

$$(\texttt{case}\ \texttt{id}_Y\ \texttt{of}\ [\, j_i \Rightarrow f_i \,]_{1 \le i \le n}) \equiv [\, j_i \Rightarrow f_i \,]_{1 \le i \le n} \ : \ Y \to Z \ .$$

The basic specifications and the basic theories form a diagrammatic logic, in this paper it is called the *basic* logic. The rules of this logic are the identity and composition rules, as in section 2, together with the rules for the existence and unicity of matches and for the extensivity of sums.

Remark 3.8 (booleans). In order to recover a type of booleans, a sum Bool = F + T can be used. Then a function with values in Bool is called a *predicate*. The inverse image of the sum Bool = F + T by a predicate $p : X \to$ Bool is also a sum, say $X = X_b + X_{\overline{b}}$, because of the extensivity property. In the basic theory of sets, it can be assumed that the types F and T are interpreted as singletons, so that Bool is interpreted as the usual set of booleans. Then, in every model M, the sets $M(X_b)$ and $M(X_{\overline{b}})$ are the parts of $M(X)$ where the map $M(b)$ is true and false, respectively.

Example 3.9 (the basic specification Σ_{nat}). The graph in example 2.5 can be considered as a basic specification, with no equation and no sum. The rules of the basic logic can be used for deriving, for instance, the functions $[s \Rightarrow s\,.\,s \mid z \Rightarrow z] : \mathrm{Nat} \to \mathrm{Nat}$, and (the subscript Nat is omitted):

$$p = \texttt{case id of } [\, s \Rightarrow \texttt{id} \mid z \Rightarrow z \,] \equiv [\, s \Rightarrow \texttt{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat}\ .$$

From its definition, the function p satisfies the equations $p\,.\,z \equiv z$ and $p\,.\,s \equiv \texttt{id}$. As in example 2.5, we are interested in the model of naturals of Σ_{nat}, called M_{nat}. In this model, the function p must be interpreted as the predecessor map $\mathrm{pred} : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{pred}(n) = n - 1$ for each positive n and $\mathrm{pred}(0) = 0$.

4 A decorated logic for exceptions

4.1 Three keywords for exceptions

We use the keywords `raise` for raising exceptions and `handle` for handling them, as in Standard ML.

The predecessor map $\mathrm{pred} : \mathbb{N} \to \mathbb{N}$ from example 3.9 can also be formalised in the following way, if some mechanism for exceptions is available:

First, an exception e is created:

$$\texttt{Exception}\ e$$

Then, a function $p' : N \to N$ is generated, such that $p'(z)$ raises the exception e:

$$p'(x) = \texttt{case}\ x\ \texttt{of}\ [\, s(y) \Rightarrow y \mid z \Rightarrow \texttt{raise}\ e \,]$$

Finally, a function $p'' : N \to N$ is generated, that calls p' and handles the exception e:

$$p''(x) = p'(x)\ \texttt{handle}\ [\, e \Rightarrow z \,]$$

The basic logic is now modified, in order to be able to deal with the mechanism of exceptions, with its three keywords:

$$\texttt{Exception},\quad \texttt{raise},\quad \texttt{handle}\,.$$

For this purpose, we use a kind of logic where the functions are *decorated*: each function is associated to a symbol, which is called its *decoration*, and which appears as a superscript. The decorations are "v" for *value* and "c" for *computation*, they are borrowed from the monads approach [11]. What is new here, is that the rules of the logic are also decorated, as will be explained below. In particular, various decorations of the extensivity property will give rise to various kinds of case distinctions, which in turn will be used for formalizing the treatment of exceptions. We claim that expressions of the form:

$$\texttt{raise}\ e \qquad \text{or} \qquad f\ \texttt{handle}\ g\,,$$

can be considered as decorated functions; the keywords "`raise`" and "`handle`" are constructors for new decorated functions, very much like "[. . .]" and "`case`" are constructors for new basic functions. Moreover, the decoration of every function can be easily derived from the use of the keyword "`Exception`" and from the rules of the decorated logic, as follows: every exception is a computation, and every function involving a computation is a computation.

One issue with the decorated logic is that it does not have set-valued models in such a simple way as the basic logic in section 3 or the explicit logic in section 5, which blurs the intuition about this logic. In section 5, the decorated logic will be mapped to the explicit logic, and a set-valued interpretation will then be recovered.

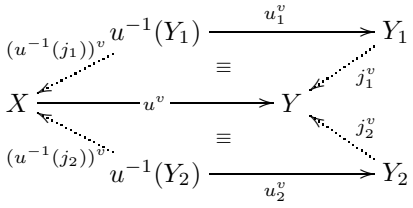
Example 4.1 (the decorated specification $\Sigma_{\mathrm{nat,deco}}$). In the next examples, a decorated specification $\Sigma_{\mathrm{nat,deco}}$ is built progressively, so that a predecessor decorated function p is defined in example 4.2 without using exceptions, then a predecessor decorated function p'' is defined in example 4.2 with the help of exceptions, and finally (also in example 4.2) it is proved, in the decorated logic, that p'' is congruent to p.

4.2 The decoration "v" for "value"

The functions that have nothing to do with the exceptions are called *values*; they are decorated with the symbol v, i.e., the notation f^v means that the function f is a value. An equation between values is called a *value equation*, i.e., the notation $f \equiv^v g$ means that $f^v \equiv g^v$ is an equation between values. The identities are values, and the composition of values is a value. The value equations generate a congruence. The sums of types behave as in the basic logic, with values instead of arbitrary functions: the coprojections are values, a match of values is a value, and the extensivity property holds for values, so that cases over values give rise to values. These sums, matches and cases are denoted as in the basic logic, in particular the initial type for values is denoted 0. For the case construction, this means that a case like "$\mathtt{case}\ u\ \mathtt{of}\ [\, j_i \Rightarrow f_i \,]_i$", where u and the f_i's are values, is the value:

$$(\mathtt{case}\ u^v\ \mathtt{of}\ [\, j_i^v \Rightarrow f_i^v \,]_{1\le i\le n})^v \;=\; (\mathtt{case}\ u\ \mathtt{of}\ [\, j_i \Rightarrow f_i \,]_{1\le i\le n})^v \;=\; [\, u^{-1}(j_i) \Rightarrow f_i \,]_{1\le i\le n}{}^v\ .$$

So, one rule of the decorated logic is the extensivity rule for values, which says that every sum has a unique inverse image by every value. For binary sums, this rule can be illustrated as follows.



Example 4.2 (the value part of $\Sigma_{\mathrm{nat,deco}}$). In our example, the value part of the decorated specification $\Sigma_{\mathrm{nat,deco}}$ is a copy of the basic specification Σ_{nat} from example 3.9. Hence, $\Sigma_{\mathrm{nat,deco}}$ has two types Unit and Nat, two values $z^v : \mathrm{Unit} \to \mathrm{Nat}$ and $s^v : \mathrm{Nat} \to \mathrm{Nat}$, and no value equation. It generates a value:

$$p^v = (\mathtt{case}\ \mathtt{id}\ \mathtt{of}\ [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow z \,])^v \equiv [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow z \,]^v : \mathrm{Nat} \to \mathrm{Nat}$$

so that $p \,.\, s \equiv^v \mathtt{id}$ and $p \,.\, z \equiv^v z$.

4.3 The decoration "c" for "computation"

All the functions that may raise exceptions are called *computations*; they are decorated with the symbol c, as well as the equations between them. Since computations *may* (and not *must*) raise exceptions, each value f^v may be *coerced* into a computation f^c, and similarly each value equation may be coerced into a computation equation. The composition of computations yields a computation, and the computation equations generate a congruence. In the composed computation $(g \,.\, f)^c = g^c \,.\, f^c$, it is expected that any exception which is raised by f^c is propagated by g^c: this is proved in theorem 4.5.

A match of computations is a computation, in a straightforward way. When $n = 0$, this means that the initial type for values is also initial for computations: for every type X, there is a unique value $[\,]_X^v : 0 \to X$, and its coercion as a computation $[\,]_X^c$ is the unique computation $[\,]_X^c : 0 \to X$.

Since a match of computations is a computation, a case like "$\mathtt{case}\ u\ \mathtt{of}\ [\, j_i \Rightarrow f_i \,]_i$" is defined when the f_i's are computations and u is a value; the same notation "$\mathtt{case}$" is used for this construction:

$$(\mathtt{case}\ u^v\ \mathtt{of}\ [\, j_i^v \Rightarrow f_i^c \,]_{1\le i\le n})^c \;=\; (\mathtt{case}\ u\ \mathtt{of}\ [\, j_i \Rightarrow f_i \,]_{1\le i\le n})^c \;=\; [\, u^{-1}(j_i) \Rightarrow f_i \,]_{1\le i\le n}{}^c\ .$$

But there is no such definition when u is a computation; indeed, if u raises an exception, there is no canonical way to decide which Y_i the exception "comes from". However, in section 4.6 a special situation is described, where some kind of "`case` u^c `of` ..." can be defined, when u is a computation.

4.4 The keyword `Exception`

In a decorated specification, the values are generated from some elementary values, which are the operation symbols of a signature, and the computations are generated from some elementary computations, which are the *exceptions*. Recall that a computation $f^c : X \to Y$ in a decorated specification may raise an exception instead of returning a result of type Y. Following this idea, we consider that a declaration "`Exception` e `of` P", for any type P, adds to the decorated specification a computation $e^c : P \to 0$: indeed, such a computation cannot return a result of type 0, since 0 stands for the empty set, hence it has to raise an exception.

$$P \xrightarrow{\quad e^c \quad} 0$$

In this paper, for simplicity, it is assumed that all the exceptions in a decorated specification are given once and for all. The exceptions form the coprojections of a new kind of sum in the decorated specification; this *exceptional sum* is studied in section 4.7.

Example 4.3 (the exception of $\Sigma_{\mathrm{nat,deco}}$). In the decorated specification $\Sigma_{\mathrm{nat,deco}}$, the declaration "`Exception` e" adds a computation $e^c : \mathrm{Unit} \to 0$, from which other computations will be derived in example 4.6.

4.5 The keyword `raise`

Recall that 0 is an initial type for values and for computations. We claim that when a function $f : X \to Y$ *raises* an exception e, this means that the exception e can be viewed as an expression of type Y. This is expressed in the following definition.

Definition 4.4 (the keyword `raise`). The keyword `raise` is the polymorphic value:

$$\mathtt{raise}_Y{}^v = [\,]_Y{}^v : 0 \longrightarrow Y \ .$$

In a decorated specification Σ, let $e^c : P \to 0$ be an exception and Y a type. To *raise the exception* e^c *in the type* Y is to build the composition:

$$(\mathtt{raise}_Y \,.\, e)^c : P \longrightarrow Y \ .$$

The following result proves that the exceptions propagate, as required; it is a consequence of the unicity of the empty sum.

Theorem 4.5 ***(propagation of exceptions).*** *For every computations $f^c : X \to 0$ and $g^c : Y \to Z$ (typically, when f^c is an exception):*

$$g \,.\, \mathtt{raise}_Y \,.\, f \equiv^c \mathtt{raise}_Z \,.\, f \ .$$

Example 4.6 (raising an exception in $\Sigma_{\mathrm{nat,deco}}$). In the decorated specification $\Sigma_{\mathrm{nat,deco}}$, the computation p' is defined as follows:

$$p'^c = (\ \mathtt{case}\ \mathtt{id}\ \mathtt{of}\ [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow \mathtt{raise} \,.\, e \,]\ )^c \equiv [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow \mathtt{raise} \,.\, e \,]^c : \mathrm{Nat} \to \mathrm{Nat}$$

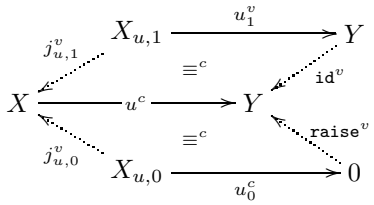
It follows from theorem 4.5 that, for every computation $g^c : \mathrm{Nat} \to \mathrm{Nat}$, the computation $(g \,.\, p' \,.\, z)^c$ raises the exception e.

4.6 The case construction over a computation

The case construction over a computation, which is described now, can be used only inside a handle construction (section 4.8). Such a construction occurs only with respect to a sum of the form $Y + 0$, for any type Y. It is easy to prove that the vertex of this sum is isomorphic to Y, with the coprojections $\mathtt{raise}_Y$ and $\mathtt{id}_Y$ (the subscript Y is often omitted): indeed, the proof involves only values, it is similar to the usual proof in the basic logic. This sum $Y = Y + 0$ may be used as the other sums, for building matches of values and matches of computations, and also for building inverse images of values, but this has little interest: the inverse image of the sum $Y = Y + 0$ by a value $u^v : X \to Y$ is simply the sum $X = X + 0$. The interesting property of the sum $Y = Y + 0$ is that there is a special rule for it: this sum has an inverse image by every *computation* $u^c : X \to Y$. Indeed, if u raises an exception, then we decide that this exception "comes from" the 0 part of the sum $Y = Y + 0$. More precisely, this inverse image is defined below.

Definition 4.7 (the inverse image of a sum by a computation). Let $u^c : X \to Y$ be a computation. An *inverse image of the sum* $Y = Y + 0$ *by the computation* u is a sum $X = X_{u,1} + X_{u,0}$, with value coprojections $j^v_{u,1} : X_{u,1} \to X$ and $j^v_{u,0} : X_{u,0} \to X$, and with a *value* $u^v_1 : X_{u,1} \to Y$ and a *computation* $u^c_0 : X_{u,0} \to 0$ such that:

$$u \,.\, j_{u,1} \equiv^c u_1 \quad \text{and} \quad u \,.\, j_{u,0} \equiv^c \mathtt{raise}_Y \,.\, u_0 \,.$$



Some properties of this inverse image are stated now, their proof is easy. The second one shows that there is no ambiguity in our definition: when a computation u^c comes, by coercion, from a value u^v, then the inverse image of $Y = Y + 0$ by the computation u is the same as the inverse image of $Y = Y + 0$ by the value u. The last property proves the "back-propagation" of the raising of exceptions, with respect to values: if $u' \,.\, u$ raises an exception, and if u is a value, then u' raises the same exception.

Proposition 4.8 ***(properties of the inverse image of a sum by a computation).***

- *Let* $u^c, u'^c : X \to Y$ *be two computations such that* $u \equiv^c u'$*, then* $u^{-1}(Y + 0) = u'^{-1}(Y + 0)$.
- *Let* $u^v : X \to Y$ *be a value, then* $(u^v)^{-1}(Y + 0) = X + 0$.
- *Let* $u^c : X \to Y$ *be a computation and* $u'^v : Y \to Z$ *a value, then* $(u'^v \,.\, u)^{-1}(Z + 0) = u^{-1}(Y + 0)$.
- *Let* $u^c : X \to Y$ *be a computation such that* $u \equiv^c \mathtt{raise}_Y \,.\, f$ *for some computation* f^c*, then* $(u^v)^{-1}(Y + 0) = 0 + X$.
- *Let* $u^c : X \to Y$ *be a computation and* $u'^v : Y \to Z$ *a value, such that* $u' \,.\, u \equiv^c \mathtt{raise}_Z \,.\, f$ *for some computation* f^c*, then* $u \equiv^c \mathtt{raise}_Y \,.\, f$*.*

Definition 4.9 (extensivity for computations). A sum $Y = Y + 0$ is *extensive for computations* if, for every computation $u^c : X \to Y$ there is an inverse image of the sum $Y = Y + 0$ by the computation u, and it is unique.

The rule of extensivity for computations states that in a decorated theory, for every type Y the sum $Y = Y + 0$ is extensive for computations.

Definition 4.10 (cases over computations). Let $u^c : X \to Y$ be a computation, and $f_1^c : X_{u,1} \to Z$ and $f_0^c : X_{u,0} \to Z$ two computations. Then the computation "$\texttt{case}^c\ u\ \texttt{of}\ [\,\texttt{id} \Rightarrow f_1 \mid \texttt{raise} \Rightarrow f_0\,]$", which is called a *case over computation* construction, is defined as:

$$(\,\texttt{case}^c\ u^c\ \texttt{of}\ [\,\texttt{id} \Rightarrow f_1 \mid \texttt{raise} \Rightarrow f_0\,]\,)^c \;=\; [\,j_{u,1} \Rightarrow f_1 \mid j_{u,0} \Rightarrow f_0\,]^c \;:\; X \to Z\ .$$

This means that the case over computation function is characterized by the equations:

$$(\texttt{case}^c\ u\ \texttt{of}\ [\,\texttt{id} \Rightarrow f_1 \mid \texttt{raise} \Rightarrow f_0\,] \,.\, (u^{-1}(j_i)) \equiv f_i \quad \text{, for } \ 1 \leq i \leq n\ .$$

4.7 The exceptional case construction

Let us come back to the declarations of exceptions. The exception declarations "$\texttt{Exception}\ e_i\ \texttt{of}\ P_i$", for $1 \leq i \leq k$, add to the decorated specification a sum of a new kind, called the *exceptional sum*, which allows to test which one among the e_i's is some given exception. From now on, let:

$$e_i^c : P_i \to 0, \text{ for } 1 \leq i \leq k\ ,$$

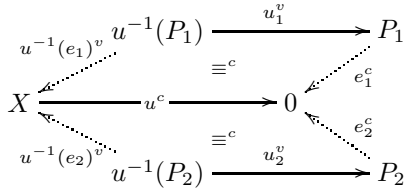
be the exceptions in some given decorated specification.

Definition 4.11 (the exceptional sum). The *exceptional sum* $0 = \sum_{i=1}^k P_i$ has vertex 0 and coprojections the computations e_i^c's for $1 \leq i \leq k$.

The exceptional sum is quite special: its coprojections are computations, instead of values, and it is used only inside a handle construction (section 4.8). The exceptional sum enjoys a decorated version of only one among the properties of sums, namely the extensivity, as follows.

Definition 4.12 (the inverse image of the exceptional sum by a computation). Let $u^c : X \to 0$ be a computation. An *inverse image of the exceptional sum by* u^c is a sum $X = \sum_{i=1}^k u^{-1}(P_i)$, with *values* coprojections $(u^{-1}(e_i)^v$, together with *values* $u_i^v : u^{-1}(P_i) \to P_i$ such that for each i:

$$u \,.\, (u^{-1}(e_i)) \equiv^c e_i \,.\, u_i\ .$$



Definition 4.13 (extensivity of the exceptional sum). The exceptional sum is *extensive* if it has a unique inverse image by every computation with type 0.

The rule of extensivity for exceptions states that in a decorated theory, the exceptional sum is extensive.

Now the exceptional case construction can be defined, as another decorated version of the basic case construction.

Definition 4.14 (exceptional cases). Let $u^c : X \to 0$ be a computation, I a subset of $\{1, \ldots, k\}$, and for each $i \in I$ let f_i^c be a computation:

$$f_i^c : u^{-1}(P_i) \to Y\ .$$

For each $i \notin I$, let f_i^c be the *default* computation:

$$f_i^c = (\mathtt{raise}_Y \,.\, u \,.\, u^{-1}(e_i))^c : u^{-1}(P_i) \to Y \ .$$

Then the computation "$\mathtt{case}^e\ u\ \mathtt{of}\ [\, e_i \Rightarrow f_i \,]_{i\in I}$", which is called an *exceptional* case construction, is defined as:

$$(\ \mathtt{case}^e\ u^c\ \mathtt{of}\ [\, e_i \Rightarrow f_i \,]_{i\in I}\ )^c\ =\ [\, u^{-1}(e_i) \Rightarrow f_i \,]_{1\le i\le k}{}^c\ :\ X \to Y\ .$$

This means that the computation "$\mathtt{case}^e\ u\ \mathtt{of}\ [\, e_i \Rightarrow f_i \,]_{i\in I}$" is characterized by the equations:

$$\mathtt{case}^e\ u\ \mathtt{of}\ [\, e_i \Rightarrow f_i \,]_{i\in I} \,.\, (u^{-1}(e_i)) \equiv f_i\ \ \text{, for }\ 1 \le i \le k\ .$$

Example 4.15 (an exceptional case in $\Sigma_{\mathrm{nat,deco}}$). In the decorated specification $\Sigma_{\mathrm{nat,deco}}$, there is only one exception $e_1 = e$, so that $k = 1$ and $P_1 = \mathrm{Unit}$, in the exceptional sum. We may consider the computations $u^c = e^c : \mathrm{Unit} \to 0$ and:

$$w^c = \mathtt{case}^e\ u\ \mathtt{of}\ [\, e \Rightarrow z \,] : \mathrm{Unit} \to \mathrm{Nat}\ .$$

Then clearly $u^{-1}(e) = \mathtt{id}_{\mathrm{Unit}}$, so that $w^c \equiv^c z : \mathrm{Unit} \to \mathrm{Nat}$.

4.8 The keyword `handle`

The keyword "`handle`" has two arguments: for instance, in the function "p' `handle` $[\, e \Rightarrow z \,]$", the arguments of `handle` are p' and $[\, e \Rightarrow z \,]$. There are two nested kinds of cases in a handling expression "f `handle` g". The first one tests whether f raises an exception, and when this is true, the second one tests which is the raised exception. The first one is a case distinction over a computation, as in section 4.6, and the second one is an exceptional case distinction, as in section 4.7. Now, the handling construction is easily defined from these two kinds of cases.

Definition 4.16 (the keyword `handle`). Let $u^c : X \to Y$ be a computation, and let $X = X_{u,1} + X_{u,0}$ be the inverse image of the sum $Y = Y + 0$ by the computation u^c, together with the restrictions $u_1^v : X_{u,1} \to Y$ and $u_0^c : X_{u,0} \to 0$. Let $X_{u,0} = \sum_{i=1}^k u_0^{-1}(P_i)$ be the inverse image of the exceptional sum by the computation u_0^c. Let I be a subset of $\{1, \ldots, k\}$ and for each i in I, let $f_i^c : u_0^{-1}(P_i) \to Y$ be a computation. To *handle an exception arising from u^c according to the match* $[\, e_i \Rightarrow f_i \,]_{i\in I}$ is to build the computation:

$$(u\ \mathtt{handle}\ [\, e_i \Rightarrow f_i \,]_{i\in I})^c = (\mathtt{case}^c\ u\ \mathtt{of}\ [\, \mathtt{id}_Y \Rightarrow u_1 \mid \mathtt{raise}_Y \Rightarrow f \,])^c : X \longrightarrow Y\ ,$$

where f is the computation:

$$f^c = (\mathtt{case}^e\ u_0\ \mathtt{of}\ [\, e_i \Rightarrow f_i \,]_{i\in I})^c : X_{u,0} \longrightarrow Y\ .$$

The following result proves that the exceptions are handled as required; it can be compared to the rules for "`handle`" in the definition of SML [10].

Theorem 4.17 ***(properties of the handling of exceptions).***

- *Let $u_1 \equiv^c u_2 : X \to Y$, then (with the above notations):*

$$u_1\ \mathtt{handle}\ [e_i \Rightarrow f_i]_{i\in I} \equiv^c u_2\ \mathtt{handle}\ [e_i \Rightarrow f_i]_{i\in I}\ .$$

- *For every value $u^v : X \to Y$:*

$$u\ \mathtt{handle}\ [e_i \Rightarrow f_i]_{i\in I} \equiv^c u\ .$$

- *For every computation* $u^c = \mathtt{raise}_Y \,.\, u' : X \to Y$ *where* $u'^c : X \to 0$*:*

$$u \;\mathtt{handle}\; [e_i \Rightarrow f_i]_{i\in I} \equiv^c \mathtt{case}^e\; u' \;\mathtt{of}\; [\, e_i \Rightarrow f_i \,]_{i\in I} \;.$$

If in addition $u' = e_j \,.\, u'' : X \to Y$ *for some* $j \in \{1, \ldots, k\}$ *and some value* $u''^v : X \to P$*, then:*

$$u \;\mathtt{handle}\; [e_i \Rightarrow f_i]_{i\in I} \equiv^c f_j \quad \textit{if} \quad j \in I \;,$$

$$u \;\mathtt{handle}\; [e_i \Rightarrow f_i]_{i\in I} \equiv^c u \quad \textit{otherwise} \quad .$$

Example 4.18 (handling an exception in $\Sigma_{\mathrm{nat,deco}}$). From example 4.2, p^v is the value:

$$p^v = \mathtt{case}\;\mathtt{id}\;\mathtt{of}\;[\, s \Rightarrow \mathtt{id} \mid z \Rightarrow z \,] \equiv [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat} \;.$$

On the other hand, from example 4.6, p'^c is the computation:

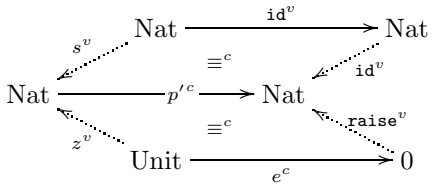
$$p'^c = \mathtt{case}\;\mathtt{id}\;\mathtt{of}\;[\, s \Rightarrow \mathtt{id} \mid z \Rightarrow \mathtt{raise}\,.\,e \,] \equiv [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow \mathtt{raise}\,.\,e \,] : \mathrm{Nat} \to \mathrm{Nat} \;.$$

Now, let:

$$p''^c = p' \;\mathtt{handle}\; [\, e \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat} \;,$$

As an example of a proof in the decorated logic, let us prove that $p'' \equiv^c p$.

It follows from the definition of p'^c that:



Hence, the inverse image of the sum $\mathrm{Nat} = \mathrm{Nat} + 0$ by the computation p' is $\mathrm{Nat} = \mathrm{Nat} + \mathrm{Unit}$, with coprojections s and z, and with ${p'_1}^v = \mathtt{id}$ and ${p'_0}^c = e$. Thus:

$$p''^c = p' \;\mathtt{handle}\; [\, e \Rightarrow z \,] \equiv^c \mathtt{case}^c\; p' \;\mathtt{of}\; [\, \mathtt{id} \Rightarrow \mathtt{id} \mid \mathtt{raise} \Rightarrow w \,] \equiv^c [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow w \,] : \mathrm{Nat} \to \mathrm{Nat} \;,$$

where, as in example 4.15:

$$w^c = \mathtt{case}^e\; e\; \mathtt{of}\; [\, e \Rightarrow z \,] \equiv^c z : \mathrm{Unit} \to \mathrm{Nat} \;.$$

It follows that:

$$p'' \equiv^c [\, s \Rightarrow \mathtt{id} \mid z \Rightarrow z \,] : \mathrm{Nat} \to \mathrm{Nat} \;.$$

Finally, from the unicity of matches, we conclude that:

$$p'' \equiv^c p \;.$$

Since p is a value, it follows that the computation p'', actually, never raises an exception.

4.9 Undecoration

Definition 4.19 (undecoration). The *undecoration* of a decorated specification Σ_{deco} is the basic specification Σ_{basic} that is obtained simply by forgetting the decorations.

In the framework of diagrammatic logics, it is easy to check that the undecoration is a morphism from the decorated logic to the basic logic.

By undecoration, every value $f^v : X \to Y$ or computation $f^c : X \to Y$ in Σ_{deco} gives rise to a function $f : X \to Y$ in Σ_{basic}. Decorated sums and cases in Σ_{deco}, give rise to ordinary sums and cases in Σ_{basic}. The sum $X = X_{u,1} + X_{u,0}$ gives rise to the sum $X = X + 0$, and the exceptional sum to a sum with vertex 0. Hence, the undecoration allows to get a simplified view on the functions and equations, by forgetting all the decorations. It allows to get a simplified view on the proofs, since the image of a proof in the decorated logic is a proof in the basic logic. This can be stated as:

"A proof in Σ_{deco} is a proof in Σ_{basic} which can be decorated".

This yields a two-step method for checking a proof in the decorated logic: first, the proof without its decorations must be valid in the basic logic, then it must be feasible to add the decorations in a way that is valid in the decorated logic.

However, this simplified view "does not preserve the meaning": for instance, when Unit is interpreted as a singleton, a constant exception $e^c : \text{Unit} \to 0$ in Σ_{deco} gives rise in Σ_{basic} to a function $e : \text{Unit} \to 0$, which has no set-valued interpretation. In section 5, the *expansion* of a decorated specification is defined; it is more subtle than the undecoration, and it "does preserve the meaning".

Example 4.20 (the undecoration of $\Sigma_{\text{nat,deco}}$). By undecorating $\Sigma_{\text{nat,deco}}$, we get a basic specification $\Sigma_{\text{nat,basic}}$, with a function $e : \text{Unit} \to 0$, so that this basic specification has no set-valued model where Unit is interpreted as a singleton. The computation p''^c in $\Sigma_{\text{nat,deco}}$, that involves the three kinds of decorated cases, gives rise in $\Sigma_{\text{nat,basic}}$ to a function that involves three times the basic case distinction.

5 A logic with explicit exceptions

5.1 Expansion

The exceptions are now considered in an *explicit* way, which means that there is a type of exceptions E which formalizes the set of exceptions, and that E appears in the type of a function, as soon as this function may raise an exception. This corresponds to the *explicit* logic, which has no decorations. It is an enrichment of the basic logic with a distinguished type E.

Definition 5.1 (explicit specification). An explicit specification is a basic specification together with a distinguished type E.

Definition 5.2 (expansion). The *expansion* of a decorated specification Σ_{deco} is the explicit specification Σ_{expl} obtained by adding the distinguished type E, keeping each value $f^v : X \to Y$ as a function $f : X \to Y$, and replacing each computation $f^c : X \to Y$ by a function $f : X \to Y + E$.

In the framework of diagrammatic logics, it is easy to check that the expansion is a morphism from the decorated logic to the explicit logic.

So, every non-exceptional sum $\sum_{i=1}^n (j_i^v : Y_i \to Y)$ in Σ_{deco} gets expanded as a sum $\sum_{i=1}^n (j_i : Y_i \to Y)$ in Σ_{expl}. The initial type 0 in Σ_{deco} gets expanded as the initial type 0 in Σ_{expl}, and the value $\texttt{raise}_Y^v = [\,]^v : 0 \to Y$ gets expanded as $[\,] : 0 \to Y$, for each type Y. In this way, the properties of sums of values in the decorated logic get satisfied by their images in the explicit logic. This includes the existence and unicity of the inverse image of any value u^v, which gets expanded as the inverse image of the function u. This also includes the property that there are matches of computations; indeed let $(f_i^c : Y_i \to Z)_{1 \le i \le n}$ be computations in Σ_{deco}, they get expanded as functions $(f_i : Y_i \to Z + E)_{1 \le i \le n}$, and the computation $[f_1 \mid \ldots \mid f_n]^c : Y \to Z$ gets expanded as the function $[f_1 \mid \ldots \mid f_n] : Y \to Z + E$.

For the cases over computations, let $u^c : X \to Y$ be a computation in Σ_{deco}, then the expansion of the inverse image of the sum $Y = Y + 0$ by the computation u^c is the inverse image of the sum $Y + E$ by the function $u : X \to Y + E$ in Σ_{expl}.

For the exceptional cases, the exceptions $e_i^c : P_i \to 0$ get expanded as $e_i : P_i \to E$. So, the expansion of the exceptional sum is the sum $E = \sum_{i=1}^{k} P_i$, with coprojections the e_i's, and the expansion of an inverse image of the exceptional sum is an inverse image of this sum.

Since the raising and handling of exceptions have been defined in terms of these decorated case constructions, they get expanded accordingly.

Example 5.3 (the expansion of $\Sigma_{\text{nat,deco}}$). Let $\Sigma_{\text{nat,expl}}$ be the expansion of $\Sigma_{\text{nat,deco}}$: it is made of a copy of Σ_{nat} from example 3.9, together with $e : \text{Unit} \to E$, which has to be a sum, which means that e has to be invertible.

5.2 Models

Let Σ_{deco} be a decorated specification, and Σ_{expl} the explicit specification obtained by expanding Σ_{deco}. Let $\mathbb{E}$ be a fixed set, called the *set of exceptions*. A *(set-valued) model of* Σ_{expl} *with set of exceptions* $\mathbb{E}$ is defined as a (set-valued) model (in the basic sense) such that the interpretation of the distinguished type E is the set $\mathbb{E}$. So, the exceptions $e_i^c : P_i \to 0$ in Σ_{deco}, that are expanded as $e_i : P_i \to E$ in Σ_{expl}, are interpreted as maps $M(e_i) : M(P_i) \to \mathbb{E}$. It follows that $\mathbb{E}$ must be the disjoint union of the $M(P_i)$'s.

It follows, as required, that the models of the expanded specifications provide a denotational semantics for the decorated logic.

Theorem 5.4 ***(soundness).*** *The deduction system of the decorated logic is sound with respect to the explicit denotational semantics.*

This means that every equation of Σ_{deco} (either between values or between computations) is interpreted as an equality in every model of Σ_{expl}. A proof of this result can be found in [6], it relies upon the fact that the decorated and the explicit logics can be formalized as diagrammatic logics, and that the expansion is a morphism between them.

Example 5.5 (the expansion of $\Sigma_{\text{nat,deco}}$). Let $\mathbb{E} = \{\varepsilon\}$. Then $\Sigma_{\text{nat,expl}}$ has a model $M_{\text{nat,expl}}$ that interprets Unit, Nat, z, and s as $\{*\}$, $\mathbb{N}$, 0 and succ, respectively, and $e : \text{Unit} \to E$ as $\varepsilon : \{*\} \to \mathbb{E}$. In this model, the computation $(\texttt{raise}_{\text{Nat}} \, . \, e)^c$ and the value z^v are interpreted respectively as ε and 0. The value p^v is interpreted as the predecessor map $\text{pred} : \mathbb{N} \to \mathbb{N}$, such that $\text{pred}(n) = n - 1$ for $n > 0$ and $\text{pred}(0) = 0$. The computation p'^c is interpreted as the map $\text{pred}' : \mathbb{N} \to \mathbb{N} + \mathbb{E}$, such that $\text{pred}'(n) = n - 1$ for $n > 0$ and $\text{pred}'(0) = \varepsilon$. And the computation p''^c is interpreted as the map $\text{pred}'' : \mathbb{N} \to \mathbb{N} + \mathbb{E}$, such that (like pred) $\text{pred}''(n) = n - 1$ for $n > 0$ and $\text{pred}''(0) = 0$.

6 Conclusion

Two logics for dealing with exceptions are presented in this paper. The decorated logic provides a deduction system, and the explicit logic provides a denotational semantics. The expansion, from the decorated logic to the explicit logic, ensures soundness.

Perspectives include the comparison of this approach with other formalizations. Another direction for future research is to use a similar approach, via morphisms of diagrammatic logics, in order to study other computational effects; in particular, the combination of various effects should run smoothly in our diagrammatic framework.

References

[1] N. Benton, J. Hughes, E. Moggi. Monads and Effects. APPSEM Summer School September 2000. LNCS 2395 (2002).

[2] G. Bernot, M. Bidoit, C. Choppy. Abstract data types with exception handling: an initial approach based on a distinction between exceptions and errors. Theoretical Computer Science 46 (1), 13–45 (1986).

[3] A. Carboni, S. Lack, R.F.C. Walters. Introduction to extensive and distributive categories, Journal of Pure and Applied Algebra 84, 145-158 (1993) .

[4] D. Duval. Diagrammatic specifications. Mathematical Structures in Computer Science 13, 857-890 (2003).

[5] D. Duval, C. Lair. Diagrammatic specifications. Rapport de recherche IMAG-LMC 1043 (2002). http://www-lmc.imag.fr/lmc-cf/Dominique.Duval/

[6] D. Duval, J.-C. Reynaud. Diagrammatic logic and effects: the example of exceptions. ccsd-00004129 (2004).

[7] D. Duval, J.-C. Reynaud. Diagrammatic logic and exceptions: an introduction. Proceedings MAP05, Mathematics, Algorithms, Proofs, Dagstuhl Seminars (2005).

[8] M. Gogolla, K. Drosten, U. W. Lipeck, H.-D. Ehrich. Algebraic and operational semantics of exceptions and errors. In Theoretical Computer Science, 6th GI-Conference. Lecture Notes in Computer Science145, Springer 141-151 (1983).

[9] J. Laird. Exceptions, continuations and macro-expressiveness. In the proceedings of the European Symposium on Programming, ESOP (2002).

[10] R. Milner, M. Tofte, R. Harper. The definition of Standard ML, MIT Press (1990).

[11] E. Moggi. Notions of computation and monads, Information and Computation 93, 55–92 (1991).

[12] G. Plotkin, J. Power. Semantics for algebraic operations. Electronic Notes in Theoretical Computer Science 45, 1–14 (2001).

[13] G. Plotkin, J. Power. Algebraic Operations and Generic Effects. Applied Categorical Structures 11 (1), 69-94 (2003).

[14] P.-Y. Schobbens. Exceptions for algebraic specifications. Science of Computer Programming 20 (1993).

[15] D. Walter, L. Schröder, T. Mossakowski Parametrized Exceptions. In CALCO 2005, Lecture Notes in Computer Science 3629, Springer 424-438 (2005).